Backward Analysis for Inferring Quantified Preconditions

Technical Report TR-2007-12-01

Tal Lev-Ami * Mooly Sagiv
Tel Aviv University

Thomas Reps
GrammaTech, Inc. and
Univ. of Wisconsin

Sumit Gulwani
Microsoft Research, Redmond, WA

Abstract

This paper presents a method to infer preconditions that contain quantifiers. Such invariants can quantify over an unbounded number of storage locations and elements of arrays, and allow shape properties of programs that manipulate pointers and dynamically allocated data structures to be described concisely. The preconditions that are found ensure that any assertions that exist in the code will hold when they are reached.

Precondition discovery is accomplished by performing backward shape analysis, with the goal of underapproximating the weakest liberal precondition. To be able to perform (underapproximating) generalization in loops, the method is based on overapproximating the complement; that is, we start with the complement C of the assertion(s) (hence C describes the states that should not be reachable) and perform overapproximating shape analysis backwards to find an overapproximation D of the states at the beginning of the procedure that could reach C. Taking the complement of D provides the desired precondition.

We concentrate our discussion on programs that are written in languages that permit destructive updating of dynamically allocated storage. Even though we are dealing with such a challenging problem, we have found that the abstractions that we use are precise enough to work backwards through loops successfully. Moreover, the use of abstraction provides significant leverage for automated backward reasoning: abstraction can lower the number of *descriptors* of aliasing possibilities that an automatic technique would otherwise have to enumerate explicitly, which can greatly lower the size of the (abstract) state space that is explored.

1. Introduction

In this paper, we use backward analysis to infer preconditions that contain quantifiers. The preconditions that we find ensure that any assertions that exist in the code will hold when they are reached. The assertions can be either programmer-supplied, or based on requirements of the programming language (e.g., no dereferences of `NULL` are permitted).

To accomplish this, we perform backward analysis. A backward analysis works counter to the program's flow of control, which has both advantages and disadvantages. For instance, an advantage of a backward analysis is that it can be demand-driven: it seems natural to work backward when the goal is to check a single assertion. The approach has the potential to determine answers faster because its work can be focused on specific points of interest.

On the other hand, backward analysis presents substantial challenges:

1. The standard approach to inferring preconditions is to start with an assertion and work backwards, computing the weakest (liberal) precondition (*wlp*) at each step. Many logics are closed under *wlp* with respect to non-looping code fragments, and this has made it a fundamental primitive in formal reasoning.

   Unfortunately, there is a stumbling block: in the presence of a loop, there is a need to find a loop invariant. When performing static program analysis, at each program point we are usually seeking an overapproximation of the states that can arise there; i.e., we want to represent at least the states that can occur at each program point, and possibly more. However, when trying to infer a precondition, this is not the case: a superfluous state at the beginning of the procedure could be one that leads to a subsequent state that violates the assertion. Thus, when inferring preconditions, an underapproximation is needed.

   However, underapproximations have problems of their own: the difficulty with working with underapproximations when trying to find a loop invariant is the difficulty in generalizing. In a setting in which overapproximation is appropriate, if we have several states that we need to cover, we can over-generalize and still get a sound result. This is not the case when working with underapproximations.
2. For a language with destructive updating, the forward semantics of the program means that information is lost when memory locations are overwritten. Consequently, the backward (abstract) semantics has to be non-deterministic—that is, for at least some backward transformers, from a given post-state will have to non-deterministically generate all possible pre-states from which the post-state could have been created.
3. The problem is exacerbated when the programming language allows destructive updates through pointers. Let s be an assignment statement of the form $v = e$. For simple assignments, the weakest liberal precondition of Q with respect to s (denoted by $wlp(s, Q)$) is obtained from Q by replacing every occurrence of v in Q by e. However, assignments through pointers (i.e., statements of the form $*p = e$) must be handled by the approach of Morris [17], where the weakest liberal precondition of Q with respect to $*p = e$ is Q with every reference r in Q replaced by "if $*p$ == $*q$ then e else r".

To overcome the problem of working with underapproximations (item 1), we use a different approach, namely, we use an overapproximation of the complement: we take the complement of the

* Supported by an Adams Fellowship through the Israel Academy of Sciences and Humanities

[Copyright notice will appear here once 'preprint' option is removed.]

assertion(s) (i.e., we start with the states that should not be reachable) and find an overapproximation of the states at the beginning of the procedure that could reach them. Taking the complement of the resulting abstract states provides the desired precondition.

It is important to understand the distinction between two different, but related, goals for a precondition-discovery method:

(1) For a given assertion instance a (or set of assertion instances A), find a precondition that ensures that each time the program reaches a (alternatively, any of the $a \in A$) in some state σ, the assertion at a holds in σ.

(2) For a given assertion instance a (or set of assertion instances A), find a precondition that ensures that (i) the program will reach a (alternatively, one of the $a \in A$), and (ii) each time the program reaches a (alternatively, any of the $a \in A$) in some state σ, the assertion at a holds in σ.

In this paper, we address version (1); that is, we do not provide the kind of "must-reach" guarantee provided by (2)(i), even for terminating runs of the program. In particular, a terminating run of the program can start in a state that satisfies the generated precondition, but, due to non-determinism in the program, bypass assertion a completely.[1]

This paper makes the following contributions:

- We develop methods that address items 2 and 3 in our list of challenges. In particular, our approach might be characterized as "Morris tamed by abstraction": the method "respects Morris" in the sense that it accounts for all of the possible aliasing combinations; however, the size of the set of explicitly enumerated situations is kept low by means of abstraction—that is, detailed information is kept only for some of the pre-structures for the transition, and others are merged into combined, less-detailed descriptors by abstraction.

  We introduce two heuristics that seem to be quite effective at reducing the number of descriptors of aliasing possibilities that have to be enumerated explicitly, yet allow interesting preconditions to be inferred successfully.
  - "Do not enumerate detailed aliasing possibilities for nodes that one is not currently interested in." (Of course, this leaves open the question of what we mean by "currently interested in"; this is addressed in §4.)
  - "Only keep track of the part of the heap that has been relevant (while working backward) so far." (Again, the concept of "relevant so far" will be explained later; see §4.2.)
- We present the basic principles underlying our technique in a form independent from the shape-analysis context, which allows the approach to be applied to perform backwards analysis and precondition discovery for a broader class of abstract domains (see §3). This represents an improvement to the state of the art for symbolic methods for abstract interpretation, because heretofore the kinds of symbolic methods that we make use of were only known for a specific class of shape-analysis domains [24]. We show that our techniques apply to programs that manipulate arrays and strings, and discuss the analysis of example programs in each area.
- We present improvements to the state of the art for symbolic methods for shape analysis [24] (see §4).
- We illustrate our approach using the example of destructive append of two lists. The `append` procedure runs down the `n`-links of the first list ("n" abbreviates "next"), and changes the `NULL`-valued `n`-link in the final cell to point to the second list. The assertions of interest are that the returned list is acyclic, and that all of the heap-allocated nodes are reachable from the head of the returned list. We show how to generate automatically a precondition that captures the condition "if the result of `append` is to be acyclic, a sufficient condition is that the two input lists are disjoint."

[1] Thus, (1) corresponds to using weakest liberal precondition—Q is the weakest liberal precondition of R with respect to S if Q is the weakest condition such that if S terminates when started in a state that satisfies Q, the state obtained after executing S satisfies R—whereas (2) would require using weakest precondition—Q is the weakest precondition of R with respect to S if Q is the weakest condition such that (i) S terminates when started in a state that satisfies Q, and (ii) the state obtained after executing S satisfies R.

```
int strlen(char *str){
[0] int i = 0;
[1] while(str[i] != '\0'){
[2]     i++;
    }
[3] return i;
}
```

Figure 1. A procedure that finds the length of a C-style null-terminated string.

Organization.. The remainder of the paper is organized as follows: §2 presents an example, and provides an overview of our goals, methods, and results obtained. §3 presents the basic principles underlying our technique in a form independent from the shape-analysis context. §4 presents the technique in the context of shape analysis, including specific strategies and solutions to specific challenges that arise in applying the method for shape analysis. §5 describes our current implementation. §6 discuses related work. §7 draws some conclusions from our work.

2. Overview

Before diving into shape analysis and the full running example of this paper, consider the `strlen` procedure of Fig. 1. The assertion in the code is implicit and states that when accessing `str[i]` at line `[1]` the index is within the allocated bounds of the string. The precondition that makes sure this assertion never fails is that the length of the allocated buffer is greater than 0 and that somewhere in the string the null character `'\0'` appears.

As is done in the C standard, we distinguish between two cases:[2] (i) i is one step after the end of the allocated buffer (a position for which we keep an auxiliary variable $|str|$), and (ii) i is completely off the buffer (i.e., $i < 0 \vee i > |str|$). We use a special unary predicate $out[str](i)$ to capture the latter case. A unary predicate $cz[str](i)$ captures the condition that the character in $str[i]$ is `'\0'`.

The negated assertion of line `[1]` can be represented by the formula $i = |str| \vee out[str](i)$. Performing backward analysis from this assertion gives us the following abstract elements (shown here as formulas) at the entry label:

- $|str| = 0$,
- $|str| = 1 \wedge \neg cz[str](0)$,
- $|str| = 2 \wedge \neg cz[str](0) \wedge \neg cz[str](1)$,
- $|str| \geq 2 \wedge \forall 0 \leq j < |str| \,.\, \neg cz[str](j)$

It is easy to see that the negation of this formula (treated as a disjunction) gives us the precondition that we are looking for.

2.1 Running Example

We use the procedure `append` shown in Fig. 2 as the running example for the paper; `append` takes two lists as arguments (pointed to by `x` and `y`). It has a loop that performs a trailing-pointer search for the final node of the `x`-list: variable `t` traverses the list until it reaches the end; variable `prev` serves as the trailing pointer, trailing `t` by one node as they move in lock-step down the list. In the

[2] In C, a pointer is permitted to point one element past the end of a buffer, but it is not allowed to be dereferenced when it is that state.

```
typedef struct node {
  int data;
  struct node *n;
} *List;
```

(a)

```
List append(List x, List y){
[0] List t = x;
[1] List prev = NULL;
[2] while(t != NULL){
[3]    prev = t;
[4]    t = t->n;
    }
[5] if(prev == NULL) x = y;
[6] else prev->n = y;
[7] assert(AcyclicList(x));
[8] assert(AllReachable(x));
[9] return x;
}
```

(b)

Figure 2. (a) Declaration of a linked-list datatype in C; (b) destructive-append procedure.

common case where x points to a non-empty list, when t becomes NULL, prev points to the final node of the list, and the n field of the final node is changed (from NULL) to point to the head node of the y-list.

Analysis goals. There are two assertions at the end of append: one on line [7],

```
[7] assert(AcyclicList(x));,
```

which asserts that x is an acyclic list, and one on line [8],

```
[8] assert(AllReachable(x));,
```

which asserts that all of the heap-allocated nodes are reachable from x. The goal of the analysis that we develop is to find a precondition that will ensure that the assertions on lines [7] and [8] will hold whenever lines [7] and [8] are reached, respectively.

There is a third, hidden assertion on line [6] due to the way that we have chosen to handle cyclicity. The analysis assumes that the original lists are acyclic, and ensures that no cycles are formed during the execution of the program by adding an extra assertion each time an operation that destructively updates a field is performed. In append, there is one such operation on line [6]: prev->n = y; the assertion checks that prev is not reachable from y.

Analysis overview. The analysis starts with the assertions and complements them. It then performs shape analysis backwards, using a variant of the Sagiv-Reps-Wilhelm (SRW) approach based on 3-valued logic [21].[3] The phase of backwards shape analysis finds an overapproximation of the states at the beginning of the program that could reach the complement of the assertions. Taking the complement of the resulting abstract states provides the desired precondition.

In shape analysis, we wish to represent a large (possibly infinite) set of memory configurations using a finite set of memory descriptors; here this is done by collapsing nodes together into "summary nodes" (drawn as double circles). We use three-valued logical structures (sets of relations in which entries for tuples are allowed to have the additional truth value $1/2$ to capture the case in which for some of the original tuples represented by the abstract tuple the value is true (1) while for others the value is false (0)).

The use of abstraction provides significant leverage for automated backward reasoning: abstraction can lower the number of *descriptors* of aliasing possibilities that an automatic technique would otherwise have to enumerate explicitly, which can greatly lower the size of the (abstract) state space that is explored. (The analysis still accounts for all of the possible aliasing combinations, but detailed information is kept only for some of the pre-structures for a given transition; others are merged into combined, less-detailed descriptors by abstraction.)

As we observed in [13], abstraction can also help in dealing with complicated properties, such as transitive closure. When done correctly, computing the transitive closure on an abstract descriptor is easier than reasoning about transitive closure in concrete stores. Thus, choosing the right abstraction can enable the analysis to reason about transitive closure with enough precision to prove desired properties.

The precondition obtained for `append`. As a way to provide intuition about what sort of results are obtained as the result of precondition discovery for append, it is actually easier to discuss the abstract states that arise from the backwards-shape-analysis phase to overapproximate the *bad states* at the entry to append. Fig. 3 shows four of the twenty-seven 3-valued structures that backwards shape analysis generates for the entry node for the assertion of acyclicity. The assertion of reachability generates an extra seventeen 3-valued structures (representing the cases in which the leak has occurred before the procedure has started).

Each shape graph shown in Fig. 3 represents a set of concrete memory configurations. Each graph typically represents lists of various lengths, with various connections among the list nodes. The shape graph shown in Fig. 3(a) represents just one memory configuration—one that consists of a three-element list in which x points to the head of the list and y points to the second node of the same list. The shape graph shown in Fig. 3(b) represents an infinite set of memory configurations, each of which is finite and contains one list structure in which x points to the head of the list, and y points to the third or later node of the same list.

Heap-allocated nodes and their properties in the represented heaps can be read off from the shape graph in the following way:

- Circles stand for abstract nodes.
  - A solid circle stands for an abstract node that represents exactly one concrete heap node. In Fig. 3(a), the circles labeled 0, 1, and 2 represent the three nodes of an input list of length three.
  - A double circle stands for an abstract node that may represent one or more concrete nodes.

    In Fig. 3(b), the double circle 2345 represents some number of nodes in the tail of the x list (but not in the tail of the y list), while the double circle 266 represents some number of nodes that are in the tail of *both* the x list and the y list).
- The name p represents pointer variable p. For instance, the names of the pointer variables of program append, namely, x, y, t, and prev appear in the shape graphs in Fig. 3.

  A dotted arrow from p to an abstract node represents the fact that p may point to some of the heap nodes represented by the abstract node, and may be false for others. Fig. 3(a), the fact that t and prev have dotted arrows to the three abstract nodes means that we have no information about which node t and prev point to, or whether either or both have the value NULL.

  The absence of a name p in a shape graph means that, in the memory configurations represented by the shape graph, program variable p definitely has the value NULL. (In Fig. 3, none of the variable names is absent.)
- A unary property q that holds for all heap nodes represented by an abstract node is represented in the graph by writing the property name q inside the node's circle.

  For example, the property "reachable-from-x-via-n", denoted in the graphs by $r[n, x]$, means that the heap nodes represented

[3] More precisely, we resurrect a slightly broader class of 3-valued logic abstractions originally used in an early version of SRW [20], and use a theorem prover as the engine for abstract interpretation, along the lines of the method presented in [24] (with improvements and some heuristics that have a substantial impact on performance). The differences from previous approaches based on 3-value logic will be explained in later sections as we explain the techniques at a deeper level.

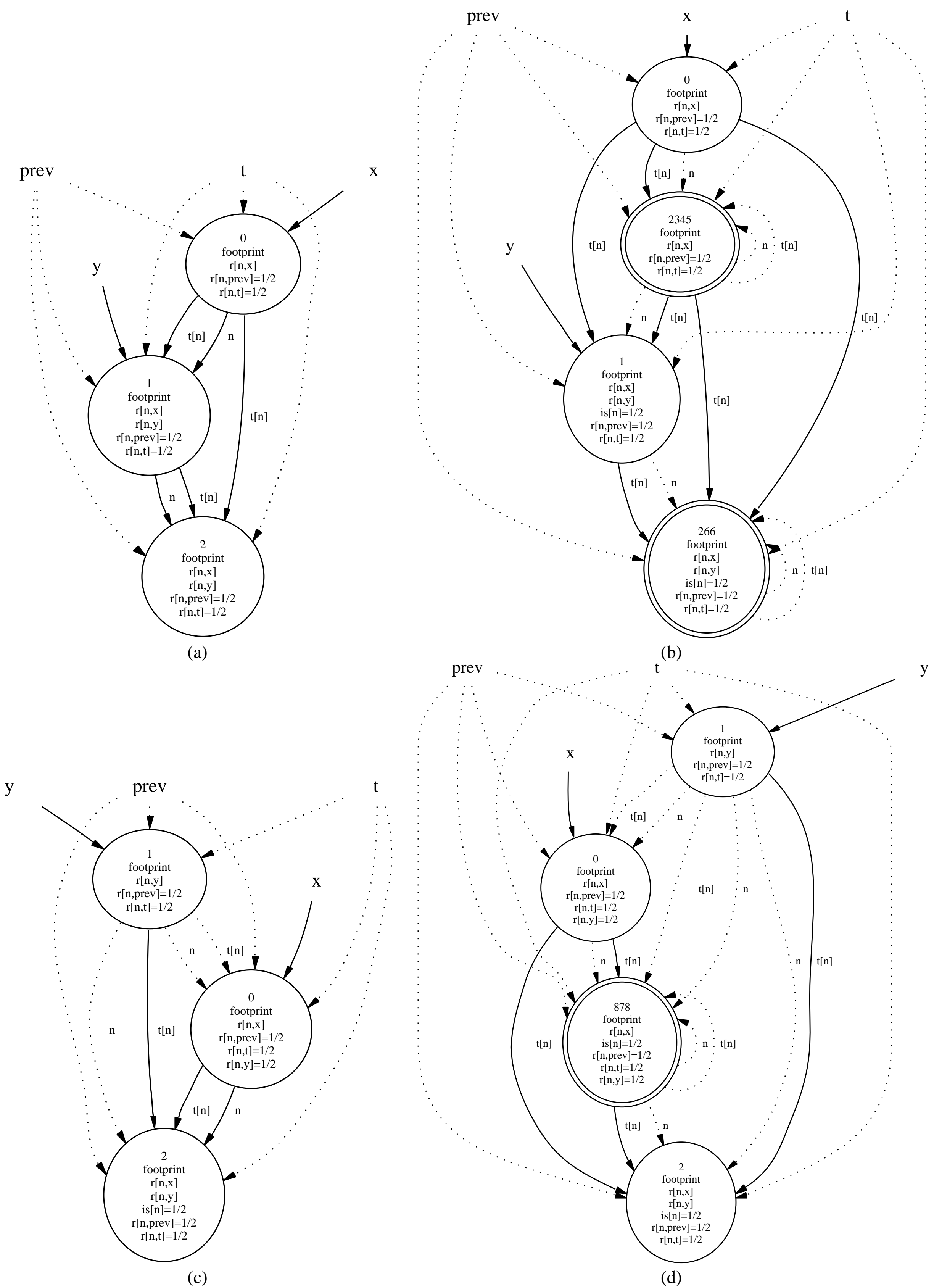

Figure 3. (a) `x` points to the head of a three-element list and `y` points to the second node of the same list; (b) `x` points to the head of a list and `y` points to the third or later node of the same list; (c) `y` points to the head of a two or three-element list that shares a common tail—possibly the whole `x`-list—with the list pointed to by `x`; (d) `y` points to the head of a list that shares a common tail—possibly the whole `x`-list—with the list pointed to by `x`.

by the corresponding abstract nodes are (transitively) reachable from pointer variable x via n-fields.

All three circles in Fig. 3(a) contain the property name $r[n, x]$. This means that all three concrete nodes represented by the three circles are reachable from x via 0 or more dereferences of n-fields.

The absence of a property name q in a circle m means that the property does not hold in any of the concrete nodes that m represents. The notation $q = 1/2$ in a circle m means that property q may hold for some of the concrete nodes that the circle represents, but that q may fail to hold for some of the concrete nodes that the circle represents.

- A solid edge labeled n between abstract nodes m and m' represents the fact that the n-field of the heap node represented by m points to the heap node represented by m'. Fig. 3(a) indicates that the n-field of the node represented by node 0 points to the node represented by node 1.
- A dotted edge labeled n between abstract nodes m and m' tells us that the n-field of one of the heap nodes represented by m may point to one of the heap nodes represented by m'. In Fig. 3(b), the dotted n edge from 1 to 2345 means that the n-field of the heap node that 1 represents may point to one of the heap nodes that 2345 represents.

  Because the analysis checks that the program can never introduce cycles, a dotted self-loop labeled n can never represent a cycle in any of the concrete stores represented.
- In general, a binary property π that holds between all the concrete nodes that abstract node m represents and all the concrete nodes that abstract node m' represents is represented by a solid edge labeled π from m to m'. Such an edge is absent if π does not hold between any pair of concrete (m, m') nodes, and dotted if π holds for some pair of concrete (m, m') nodes and does not hold for others.

  For instance, the property $t[n](v_1, v_2)$ means "Is v_2 reachable from v_1 along zero or more n-links?" In Fig. 3(b) the solid $t[n]$ edges from 0 to 2345, from 2345 to 1, and from 1 to 266 mean that the four abstract nodes represent a linked list of length four or more.

In summary, the shape graphs portray information of three kinds:

solid meaning "must hold" for properties,

absent meaning "must not hold" for properties, and

dotted, or (in the case of circles) doubled meaning "don't know" for properties.

The two other structures shown in Fig. 3 can be explained similarly.

- In Fig. 3(c), y points to the head of a two or three-element list that shares a common tail (possibly the whole x-list) with the list pointed to by x.
- In Fig. 3(d), y points to the head of a list that shares a common tail (possibly the whole x-list) with the list pointed to by x.

In each of the four cases shown in Fig. 3, if append is called with actual parameters x and y, append will introduce an n-link that connects the last node of the x-list to the node that y points to, thereby creating a cycle.

The memory configurations allowed by the precondition are the structures *not* represented by the four structures shown in Fig. 3 and their twenty-three brethren. In high-level terms, the precondition discovered captures the condition "if the result of append is to be acyclic, a sufficient condition is that the two input lists are disjoint", or, as a formula, $\forall v . \neg n^*(x, v) \vee \neg n^*(y, v)$.

3. Inferring Preconditions

The approach to precondition discovery taken in this paper is to compute a backward overapproximation of the bad states: We take the complement of the assertion(s) (i.e., the states that should not be reachable) and find an overapproximation of the states at the beginning of the procedure that can reach them. The complement of the resulting set of abstract states is the desired precondition.

A program is a transition system defined in terms of a set of relations $\{\tau_i\}$, where each τ is a relation on the set *Store* of stores. The weakest liberal precondition is characterized by the concrete operator $\widetilde{\mathrm{Pre}}[\tau]$:

$$\widetilde{\mathrm{Pre}}[\tau](S) \stackrel{\text{def}}{=} \{s \in \mathit{Store} \mid \forall s' \in \mathit{Store} : (s, s') \in \tau \Rightarrow s' \in S\}.$$

Our goal is to underapproximate $\widetilde{\mathrm{Pre}}[\tau]$ by some operator $\widetilde{\mathrm{Pre}}^{\flat}[\tau]$.

Also relevant to the discussion is the operator $\mathrm{Pre}[\tau]$:

$$\mathrm{Pre}[\tau](S) \stackrel{\text{def}}{=} \{s \in \mathit{Store} \mid \exists s' \in \mathit{Store} : (s, s') \in \tau \wedge s' \in S\},$$

which is related to $\widetilde{\mathrm{Pre}}[\tau]$ as follows:

$$\widetilde{\mathrm{Pre}}[\tau](S) = \overline{\mathrm{Pre}[\tau](\overline{S})}$$

Therefore, to underapproximate $\widetilde{\mathrm{Pre}}[\tau](S)$, we can use

$$\widetilde{\mathrm{Pre}}^{\flat}[\tau](S) \stackrel{\text{def}}{=} \overline{\mathrm{Pre}^{\sharp}[\tau](\overline{S})},$$

where $\mathrm{Pre}^{\sharp}[\tau]$ is any overapproximation of $\mathrm{Pre}[\tau]$. Thus, a suitable operator $\widetilde{\mathrm{Pre}}^{\flat}[\tau]$ can be obtained by using an operator $\mathrm{Pre}^{\sharp}[\tau]$ obtained with standard abstract-interpretation techniques.

In [5], it is shown that, under certain reasonable conditions, it is possible to give a *specification* of the most-precise overapproximation to $\mathrm{Pre}[\tau]$ for a given abstract domain. For a Galois connection defined by abstraction function α and concretization function γ, the most-precise abstract transformer that overapproximates $\mathrm{Pre}[\tau]$, denoted by $\mathrm{Pre}^{\sharp}_{\mathit{best}}[\tau]$, can be expressed as follows:

$$\mathrm{Pre}^{\sharp}_{\mathit{best}}[\tau] = \alpha \circ \mathrm{Pre}[\tau] \circ \gamma. \quad (1)$$

This defines the limit of precision obtainable using a given abstraction. An abstract transformer is sound (for an overapproximating backward analysis) if it overapproximates ($\sqsupseteq$) $\mathrm{Pre}^{\sharp}_{\mathit{best}}[\tau]$. However, Eq. (1) is non-constructive; it does not provide an *algorithm* for finding or applying $\mathrm{Pre}^{\sharp}_{\mathit{best}}[\tau]$. (Note that the above discussion applies regardless of whether τ refers to the transition relation for a single statement, or to the transition relation for the entire procedure.)

Graf and Saïdi [9] showed that, by working symbolically, decision procedures and theorem provers can be used to implement an algorithm that achieves the same result as applying the most-precise abstract transformer, for *predicate-abstraction domains* (i.e., abstract domains that are finite products of predicates on *Store*). Subsequently, we showed how to create algorithms for applying the most-precise abstract transformer for domains other than predicate-abstraction domains [19, 24]. All of these algorithms exhibit an interesting interplay between abstract elements of an abstract domain and formulas in a logic. Such ideas are pursued further in the present paper (where the logic is first-order logic with transitive closure). This paper makes a number of contributions that go beyond the work of Yorsh et al. [24]; these ideas are presented in §3.2.1, §4.2, and §4.3.

We make use of the following operations:

- The operation $\widehat{\gamma}(a)$ expresses the concretization of abstract value a as a logical formula, which sidesteps the fact that, in general, the result of applying γ to a is an infinite set of concrete stores. $\widehat{\gamma}(a)$ converts a into a formula that characterizes $\gamma(a)$ exactly; that is, the set of concrete stores that satisfy $\widehat{\gamma}(a)$ are exactly those in $\gamma(a)$.
- The operation $\widehat{\alpha}(\varphi)$ identifies the most-precise abstract value of a given abstract domain that overapproximates a set of concrete stores that are represented symbolically by formula φ.

```
procedure INFERPRECONDITION(cfg, assertions)
    foreach label in cfg do
        solution[label] := α̂(¬assersions[label])
    od
    Perform overapproximating backward analysis, starting from solution,
        until a fixpoint is reached
    return ¬γ̂(solution[cfg.entry])
end
```

Figure 4. Pseudo-code for the algorithm for inferring preconditions.

It can also be interesting to make use of these primitives, yet—for pragmatic reasons—be satisfied with something that is not as precise. In that case, $\widehat{\gamma}(S)$ should overapproximate the states represented by S.

The algorithm for inferring preconditions is given in Fig. 4. The source of the preconditions are all of the assertions in the procedure, either provided explicitly by the programmer, or implicit from the semantics of the programming language (e.g., no null-dereferences). The initial abstract element for each program point (or label) is an overapproximation of the negated assertions in that state ($\widehat{\alpha}(\neg \mathit{assersions}[\mathit{label}])$); $\widehat{\alpha}$ converts the formula of the negated assertions to a set of abstract elements that overapproximate the bad states. (Note that in this step $\widehat{\alpha}$ is not required to operate on an arbitrary formula, just on negated formulas from the assertion language used. This is discussed in §4.2.1.)

We then perform backward analysis (using chaotic iteration, based on the method for applying transformers symbolically that is described in §3.1), resulting in an overapproximation of the states at the entry point of the procedure that can reach the bad states. When a fixpoint is reached, the $\widehat{\gamma}$ operation is used to convert the set of abstract elements at the entry point of the procedure back into a formula. The precondition for the procedure is merely the negation of that formula ($\neg\widehat{\gamma}(\mathit{solution}[\mathit{cfg.entry}])$).

EXAMPLE 3.1. *It is instructive to examine how the backward analysis of `strlen` presented in §2 progresses towards the fixpoint. Consider the abstract elements at label `[1]`, i.e., the loop header. We consider the case in which $|str| = i$; the case in which $out[str](i)$ holds is handled similarly. Again, we discuss the abstract elements in terms of formulas. The initial $\widehat{\alpha}$ used (see §4.2.1) returns two abstract elements:*

- $i = |str| \wedge |str| = 0$,
- $i = |str| \wedge |str| > 0$

After one iteration through the loop, the first case is removed (because it would mean that $str[-1]$ was accessed, which is handled by the case $out[str](i)$). The second case is further split into two:

- $i = 0 \wedge |str| = 1 \wedge \neg cz[str](0)$,
- $i = |str| - 1 \wedge |str| > 1 \wedge \neg cz[str](i)$

Similarly, the following iteration advances one more step toward the beginning of the string, discarding the first case and splitting the second.

- $i = 0 \wedge |str| = 2 \wedge \neg cz[str](0) \wedge \neg cz[str](1)$,
- $i = |str| - 2 \wedge |str| > 2 \wedge \neg cz[str](i) \wedge \neg cz[str](i+1)$

In the next iteration, the abstract domain can no longer represent the specific differences between i and $|str|$ and is forced to generalize, yielding:

- $i = 0 \wedge |str| \geq 2 \wedge \forall 0 \leq j < |str| \,.\, \neg cz[str](j)$
- $0 < i < |str| \wedge |str| > 2 \wedge \forall i \leq j < |str| \,.\, \neg cz[str](j)$

The last iteration computes exactly the same abstract elements and the analysis terminates.

3.1 Applying Transformers Symbolically

In this section, we present a general algorithm for applying transformers using a theorem prover. We assume that the abstract domain used is a powerset lattice (possibly with a Hoare order—i.e., the elements of the sets have a partial order between them). We will refer to the sets in the lattice as abstract sets and to the individual elements in the sets as abstract elements. For a given transformer application, a *concrete store pair* is a pair of concrete stores such that the concrete execution of the transformer on the first store may yield the second store. We call the first component of such a pair a pre-store and second component a post-store.

The algorithm takes a transformer and an abstract set representing the post-stores. The algorithm returns an abstract set that is an overapproximation of the pre-stores that match the post-stores represented by the input abstract set, and hence is an overapproximation of $\mathrm{Pre}^{\sharp}_{\mathit{best}}[\tau]$. The algorithm is written to apply transformers backwards, but could be trivially modified to apply transformers forwards (i.e., receive as input an abstract set that represents the pre-stores and return an abstract set that represents the post-stores).

The basis of the algorithm is the use of two-vocabulary abstract elements, i.e., abstract elements that directly represent pairs of concrete stores. A two-vocabulary abstract element can be thought of as approximating the relation τ when restricted to the post-stores represented by the input abstract element. We assume that each transformer τ has a formula φ_τ that captures the meaning of the transformer by relating the possible pairs of pre- and post-stores.

The algorithm operates on one abstract element at a time; it first applies the operation EXTENDPRE, which takes an abstract element and extends it to a two-vocabulary abstract element that assumes nothing about the pre-store. Thereafter, it repeatedly applies semantic reductions [5] (i.e., operations that move to a different element in the abstract lattice without changing the set of represented concrete stores). When no more semantic reductions can be applied, it projects out the post-store vocabulary, leaving just the pre-store vocabulary.

The algorithm given in Fig. 5 is based on a division of labor between two operations.

- The SHARPEN operation performs a semantic reduction that returns a tighter overapproximation of the concrete store pairs that the two-vocabulary abstract element represents—without moving to a *set* of two-vocabulary abstract elements. SHARPEN can also discover that the two-vocabulary abstract element does not represent any concrete store pairs at all. In that case, it returns a special element $\ominus$, which causes the algorithm to give up on the current two-vocabulary abstract element and move on to the next one.
- The FOCUS operation is a semantic reduction that performs case splits by replacing a two-vocabulary abstract element by a set of two-vocabulary abstract elements that represent the same concrete store pairs. When FOCUS can perform no more case splits, it simply returns its input two-vocabulary abstract element.

The algorithm uses the SHARPEN operation to tighten the overapproximation as much as possible, and then applies FOCUS to perform case splits to further improve the precision. These are iterated to increasingly tighten the overapproximation. To ensure termination of the algorithm, we require FOCUS to return a bounded number of two-vocabulary abstract elements and perform a bounded number of case splits.

Finally, we use the PROJECTPRE operation to project the two-vocabulary abstract element onto its first (pre-store) component.

An algorithm for the SHARPEN operation is given in Fig. 6. The algorithm requires that the abstract domain provide, in addition to $\widehat{\gamma}$, the following two operations:

- The ASSERTIONS operation returns a finite set of formulas, in some limited language, that describe the possible ways in which the abstract element can be restricted (i.e., to move down in the ordering on abstract elements).

```
procedure BackwardTransform(XS_post, τ)
    result := ⊥
    foreach S_post ∈ XS_post do
        active := ExtendPre(S_post)
        while active ≠ ∅ do
            select and remove an element S from active
            S := Sharpen(S, τ)
            if S = ⊖ then skip to the next iteration
            next := Focus(S, τ)
            if next = {S} then
                result := result ⊔ ProjectPre(S)
            else
                active := active ∪ next
        od
    od
    return result
end
```

Figure 5. An abstract algorithm for applying a transformer backwards.

```
procedure Sharpen(S, τ)
    if Prover(¬(γ̂(S) ∧ φ_τ)) returns valid then return ⊖
    foreach ψ ∈ Assertions(S) do
        if Prover(γ̂(S) ∧ φ_τ → ψ) returns valid then
            S := Impose(S, ψ)
    od
    return S
end
```

Figure 6. A semantic reduction for tightening a two-vocabulary abstract element.

- The IMPOSE operation takes an assertion and returns a new abstract element that incorporates the fact that the assertion holds.

The first step of the algorithm is to check that the two-vocabulary abstract element represents at least one pair of concrete stores. This is done by checking validity of the negated $\widehat{\gamma}$ formula using a theorem prover. If the negated formula is valid, there can be no concrete store pair represented by this two-vocabulary abstract element, and the special indicator $\ominus$ is returned. Afterwards, the SHARPEN algorithm simply enumerates the assertions returned by the ASSERTIONS operation, and for each of them uses the theorem prover to check whether they are implied by $\widehat{\gamma}$ of the two-vocabulary abstract element, conjoined with the transformer formula; that is, it checks symbolically that for every pair of concrete pre- and post-stores of the transformer—as constrained by the two-vocabulary abstract element—the assertion holds. If this is the case, SHARPEN uses the IMPOSE operation to incorporate that information into the two-vocabulary abstract element.

3.2 Improvements

3.2.1 Using Model Finders

One of the main problems with working with a theorem prover to implement SHARPEN is that many provers are very good at proving validity, but very bad at proving invalidity (this is reasonable when considering the semi-decidable nature of first-order logic). When using such a theorem prover, most of the time is spent on trying to prove invalid assertions about the abstract store.

If the theory used for the abstract domain and the transformers has an effective model finder (e.g., a bounded model finder for first-order logic, such as Paradox [4]), we can do better (heuristically). The model finder has to be sound, but not necessarily complete, i.e., every model has to satisfy the formula, but there may be a satisfiable formula for which it does not produce a model. The result returned by the model finder is used to prune assertions before they are given to the theorem prover. If evaluating an assertion ψ on such a model yields `false`, then necessarily the assertion is not implied by the abstract element. A single model can be used to rule out many of the assertions. Furthermore, any time we pass the formula $\Phi \rightarrow \psi$ to the prover, we can pass $\Phi \wedge \neg\psi$ to the model finder and possibly accumulate more models.

The use of the models is especially important because of the presence of the FOCUS operation. When considering the abstract elements resulting from FOCUS, one has to consider many of the assertions that we have not been able to prove on the original abstract element. This repeated work can become a major bottleneck in the algorithm. However, if we have models for the original abstract element, we can use them to prune many of these repeated calls. Note that a model for the abstract element before FOCUS is necessarily a model for at least *one* of the abstract elements in the result (because FOCUS is a semantic reduction). However, it is usually the case that it is not a model for *all* of the returned abstract elements. We can easily check which of the abstract elements the model belongs to by evaluating the appropriate $\widehat{\gamma}$ formula on the model.

The use of bounded model finders for first-order logic has another advantage when applied to formulas with transitive closure. Because the model size is bounded, the transitive closure of a formula can be accurately formulated by using auxiliary binary predicates. These predicates iteratively represent paths of increasing length. For each j that is a power of 2, we add a predicate p_j that represents all paths of length less or equal to j. The property is recursively defined using the formulas of the form

$$\forall v_1, v_2 \,.\, p_{2\cdot i}(v_1, v_2) \leftrightarrow p_1(v_1, v_2) \vee \exists w \,.\, p_i(v_1, w) \wedge p_i(w, v_2)$$

The bounded model size gives us a bound on the path length needed. Thus, we add a logarithmic number of new predicates.

3.2.2 Combining Forward Analysis

Backward abstract interpretation can be quite expensive for sufficiently expressive domains. A possible way to improve the performance and precision of the analysis is due to Cousot and Cousot [6]: combine forward analysis and backward analysis.

The use of forward analysis can help in two ways: (1) provide information about the possible behaviors of the code, and thereby allow the backward analysis to explore fewer cases, and (2) discover case distinctions that are important and can thus be employed by the backward analysis. Of course, because we do not have a good initial condition to start with, a forward analysis of the same precision can be quite expensive. However, using a weaker analysis can still prune many of the stores for the backward analysis, greatly improving performance, and, in some cases, even the precision. Because we are symbolically computing the transformers, the information from the forward analysis can be easily supplied to the backward analysis by simply conjoining to the transformer formula the result of applying $\widehat{\gamma}$ to the value for the pre-store from the forward analysis.

In [6], the combination is taken one step further, and forward and backward analyses are iterated to improve the precision of the result. We can employ a similar idea in this context by applying the following steps. First, perform backward analysis until a fixpoint is reached and an abstract set is associated with the entry label. Any store that may cause the assertion to fail is represented by this abstract set. Thus, we can now start a forward analysis (of the same expressiveness as the backward analysis) with that set as an initial value. When the forward analysis reaches a fixpoint, the abstract sets that failed an assertion are a tighter overapproximation of the possible bad states, and thus they can be used as the new initial values for another iteration of backward analysis. The process can be iterated to further improve the precision of the abstract values computed.

3.2.3 Improving Performance

Another useful observation is that because the analysis starts from all the negated assertions, when doing the backward analysis the algorithm can assume that the assertions hold.

Finally, when working with a theorem prover, the algorithm checks whether the formula that characterizes the abstract structure implies any of the assertions. This can be done sequentially; however, there is work on theorem proving that enables a theorem prover to accept multiple conjectures in parallel and reason about them simultaneously [15], thus saving much of the redundant work of the separate proofs.

4. Inferring Preconditions for Shape Analysis

In this section, we present a method for inferring preconditions for heap-manipulating programs. The method is based on the algorithm in §3. The abstract domain used is a variation of the Canonical Abstraction domain introduced in [21].

As explained in §1 the use of destructive updates complicates backward analysis because all of the possible aliases between the pointer through which the destructive update is performed and the rest of the store need to considered. We try to alleviate this problem by grouping heap cells together (into summary nodes) and uniformly treating aliases with any member of the same summary node.

Following the approach of [21], we use logical structures to represent the concrete stores of the program, and use first-order logic with transitive closure (FO(TC)) to specify the concrete transformers. This provides great flexibility in what programming-language constructs the method can handle. For the purpose of this paper, we assume that the vocabulary used is fixed and always contains equality. Furthermore, we assume that the transformer cannot change the universe of the concrete store. Allocation and deallocation can be easily modeled by using a designated unary predicate which holds for the allocated heap cells. Similarly, we assume that the universe of the concrete store is non-empty. The abstract elements are conveniently represented as finite 3-valued logical structures. We shall explain the meaning of such an abstract element S by describing the formula $\widehat{\gamma}(S)$ to which it corresponds.

The individuals of a 3-valued logical structure are called abstract nodes. We use an auxiliary unary predicate for each abstract node to capture the concrete nodes that are mapped to it. For an abstract structure with universe $\{node_1, \ldots, node_n\}$ let $\{a_1, \ldots a_n\}$ be the corresponding unary predicates. For each k-ary predicate p in the vocabulary, each k-tuple $\langle node_1, \ldots, node_k\rangle$ in the abstract structure (called an abstract tuple) can have one of the following truth values $\{0, 1, \frac{1}{2}\}$. The truth value 1 means that the predicate p universally holds for all of the concrete nodes mapped to this abstract node, i.e.,

$$\forall v_1, \ldots, v_k \,.\, a_1(v_1) \wedge \ldots \wedge a_k(v_k) \rightarrow p(v_1, \ldots, v_k) \qquad (2)$$

Similarly the truth value 0 means that the predicate p universally does not hold, i.e.,

$$\forall v_1, \ldots, v_k \,.\, a_1(v_1) \wedge \ldots \wedge a_k(v_k) \rightarrow \neg p(v_1, \ldots, v_k) \qquad (3)$$

The truth value $\frac{1}{2}$ means that we have no information about this abstract tuple, and thus the value of the predicate p is not restricted.

We use a designated set of unary predicates called *abstraction predicates* to control the distinctions among concrete nodes that can be made in an abstract element, which also places a bound on the size of abstract elements. For each abstract node $node_i$, A_i denotes the set of abstraction predicates for which $node_i$ has the truth value 1, and $\overline{A}_i$ denotes the set of abstraction predicates for which $node_i$ has the truth value 0. Each abstract node is uniquely determined by the values of A_i and $\overline{A}_i$; i.e., every pair of abstract nodes have to differ by the value of at least one abstraction predicate. Because the number of abstraction predicates is fixed, this yields a bounded number of possible abstract nodes. The main difference between the abstract domain described here and the one used in [21] is that in [21] every pair $node_i, node_j$ of different abstract nodes either $A_i \cap \overline{A}_j \neq \emptyset$ or $\overline{A}_i \cap A_j \neq \emptyset$. This ensures that each concrete node can be represented by at most one abstract node. Here this is not the case; every pair of abstract nodes must differ by the value of at least one abstraction predicate, but the difference can be 0 versus 1, 0 versus $\frac{1}{2}$, or 1 versus $\frac{1}{2}$. Thus, in some cases there can be several ways to map the concrete nodes to the abstract nodes.

There are two additional requirements on abstract nodes. First, each node must represent at least one concrete node, i.e., $\exists v \,.\, a_i(v)$. Second, the abstract nodes in the structure represent all the concrete nodes, i.e., $\forall v \,.\, \bigvee_i a_i(v)$. Finally, to ensure that a concrete node is not simultaneously mapped to two abstract nodes, we require that $\bigwedge_{i \neq j} \forall v \,.\, \neg(a_i(v) \wedge a_j(v))$.

The vocabulary may contain additional predicates called *derived predicates*, which are explicitly defined from other predicates using a formula in first-order logic with transitive closure. These derived predicates help the precision of the analysis by recording correlations not captured by the universal information. Some of the unary derived predicates may also be abstraction predicates, and thus can induce finer-granularity abstract nodes.

We say that $S_1 \sqsubseteq S_2$ if there is a total mapping m between the abstract nodes of S_1 and the abstract nodes of S_2 such that S_2 represents all of the concrete stores that S_1 represents when considering each abstract node of S_2 as a union of the abstract nodes of S_1 mapped to it by m. Formally, $\widehat{\gamma}(S_1) \wedge \psi_m \rightarrow \widehat{\gamma}(S_2)$ where

$$\psi_m = \bigwedge_{\substack{node_i \in S_1 \\ m(node_i) = node'_j}} \forall v \,.\, a_i(v) \rightarrow a'_j(v)$$

The order is extended to sets using the induced Hoare order (i.e., $XS_1 \sqsubseteq XS_2$ if for each element $S_1 \in XS_1$ there exists an element $S_2 \in XS_2$ such that $S_1 \sqsubseteq S_2$).

The abstract domain can easily handle two-vocabulary elements by considering the predicates in the pre- and post-stores together. To avoid a name clash, we use primed versions of the predicates in the post-store. The transformer formulas can be easily written using first-order logic over this shared vocabulary. Note that the transformer formula should relate primed and unprimed versions of all the non-derived predicates. On the other hand, it is not necessary to specify the update in the derived predicates as it can be extracted from their defining formula.

4.1 Applying Transformers

To apply the abstract transformers for this domain, we use the algorithm in Fig. 5. The abstract operations are instantiated as follows.

- EXTENDPRE is performed by renaming the predicates in the post-store formula to their primed versions and using it for the two-vocabulary structure. Because we assume that no transformer can change the universe of the concrete nodes, this corresponds to assuming nothing about the pre-store.
- For PROJECTPRE, we throw away all the tuple formulas containing primed predicates (this corresponds to projecting out the primed vocabulary in the 3-valued structure). This may cause several nodes to have the same values on all the abstraction predicates. In this case, they are merged by using the same abstract node for all of them. The appropriate abstract tuples are also merged by only keeping universal formulas that still hold after the merge (i.e., that have the same truth value for all of the abstract tuples mapped to the same abstract tuple).

- The ASSERTIONS operation takes every abstract tuple whose truth value is $\frac{1}{2}$ and returns two formulas, one to represent the case in which the truth value is 0 and another to represent the case in which the truth value is 1.
- IMPOSE conjoins the assertion with the current formula.

To explain the FOCUS operation, we first define the concept of focusing a unary predicate on an abstract node. Let p be a unary predicate and $node_i$ be an abstract node such that the truth value of p on $node_i$ is $\frac{1}{2}$. In this case we return the following abstract elements:

- An element in which the value of p on $node_i$ is changed to 0.
- An element in which the value of p on $node_i$ is changed to 1.
- If $node_i$ is a summary node it can represent more than one concrete node. Thus, we return a third element in which $node_i$ is bifurcated into two nodes $node'_1$ and $node'_0$; all the abstract tuples which contain $node_i$ are duplicated to reflect that they hold on both $node'_0$ and $node'_1$. In addition, the value of p on $node'_0$ is changed to 0, and the value of p on $node'_1$ is changed to 1.

It is easy to see that the three abstract elements represent all the concrete stores that the original abstract element represented. FOCUS chooses a single abstract node and a single abstraction predicate, and performs the aforementioned operation on it. Alternatively, FOCUS returns the original element. FOCUS is thus guaranteed to return a bounded number of structures (up to three). Furthermore, because the condition is maintained that each pair of abstract nodes has an abstraction predicate with different values, there can be only a bounded number of times that FOCUS returns more than one element. Thus, the BACKWARDTRANSFORM algorithm is guaranteed to terminate.

4.2 Focus Strategy

In the context of shape analysis, one of the challenges of applying the approach of overapproximating the complement is that the representation of all the bad stores is much less compact than the representation of the good stores. Consequently, one of the important requirements for making the analysis possible is to have a strategy for concentrating on the important parts of the heap. We chose a strategy based on the heap cells that are accessed during the execution of the code. This is similar in spirit to the footprint analysis of [3].

- We introduce an auxiliary predicate called *footprint*.
- FOCUS performs case splits only according to the predicates that represent the program variables participating in the current operation. Each time such a program variable is determined to point to some node, we annotate this node using the footprint predicate.
- The true power of the footprint predicate comes by changing the PROJECTPRE operation. There, if there is a non-footprint abstract node $node_i$ and a footprint node $node_j$ such that $A_j \subseteq A_i$ and $\overline{A}_j \subseteq \overline{A}_i$, we remove the footprint annotation from $node_j$ and merge it with $node_i$. This ensures that a footprint node remains separate only as long as it has some extra information regarding the abstraction predicates.

4.2.1 Computing $\widehat{\alpha}$ of the Negated Program Assertions

The algorithm in Fig. 5 can be easily modified to compute $\widehat{\alpha}$ as well (by considering single vocabulary and not two-vocabulary elements; see also [24]). However, the algorithm presented here returns a more compact representation of the negated assertions when a restriction is placed on the form in which assertions are allowed to written. In particular, allow the programmer to write assertions in two forms. Universal or existential clauses (i.e., universally or existentially quantified disjunctions of literals).

As explained in §3, $\widehat{\alpha}$ should overapproximate the negated assertions of the program. For existential clauses this means to represent all concrete stores in which the conjunction of negated literals holds universally. Thus, an abstract element with a single abstract node $node_1$ is used. The only abstract tuples that are not $\frac{1}{2}$ are the ones that correspond to the negated literals.

For universal clauses, the negation gives an existentially quantified conjunction of negated literals. Let $v_1, \ldots, v_k$ be the existentially quantified variables. The abstract elements to represent the negated assertion are built in 3 steps:

1. Generate an abstract element for each possible aliasing between $v_1, \ldots, v_k$. Each set of aliased variables is represented by a single non-summary abstract node. These nodes are marked using the *footprint* predicate to indicate they are of interest.
2. For each abstract element, the abstract tuples that correspond to the negated literals are given the appropriate value (0 in case the predicate is now negative and 1 in case it is positive). All other abstract tuples are given the value $\frac{1}{2}$.
3. Finally, for each abstract element two cases are considered: (1) The concrete store represented contains only nodes corresponding to these abstract nodes, and (2) There are other concrete nodes that need to be represented. This is done by adding another abstract node for which all abstract tuples containing it are $\frac{1}{2}$. For this node *footprint* is 0, to indicate the nodes there are of no immediate interest.

4.3 Handling Transitive Closure

The main difficulty with using existing theorem provers to implement the given algorithm is the use of transitive closure in the defining formulas of derived predicates. Using transitive closure is crucial to the precision of the analysis. However, theorem provers do not exist for FO(TC). This is for a good reason because FO(TC) is undecidable, and transitive closure does not even have an R.E. axiomatization in first-order logic. We chose the approach of approximating the behavior of transitive closure based on [13]. In that method, a partial axiomatization of transitive closure in first-order logic is incrementally given to the theorem prover. The axiomatization is sound but incomplete. Thus, if the theorem prover stores that the formula is valid, it is also valid in FO(TC). However, it can claim that a formula is invalid even though it is valid in FO(TC). In the context of the algorithm presented here, this means that the algorithm will overapproximate the pre-stores of the transformer (which is what we want).

We allow transitive closure of only simple binary predicates. Transitive closure of more complicated formulas can be modeled by adding a new derived binary predicate whose defining formula is the formula for which transitive closure should be computed.

For each binary predicate p for which transitive closure needs to be computed, we add a new binary predicate p_{tc}. This predicate is not necessarily part of the vocabulary of the abstract domain. If this is the case, the predicate is local to the computation of the transformer. Any information collected for this predicate that is not representable in the abstract element will be forgotten in the pre-store and may have to be recomputed when the next transformer is applied. The basic axiomatization of p_{tc} are the formulas that relate it to p, i.e.,

$$\forall v_1, v_2 \,.\, p_{tc}(v_1, v_2) \leftrightarrow p(v_1, v_2) \vee \exists w \,.\, p(v_1, w) \wedge p_{tc}(w, v_2) \quad (4)$$

$$\forall v_1, v_2 \,.\, p_{tc}(v_1, v_2) \leftrightarrow p(v_1, v_2) \vee \exists w \,.\, p_{tc}(v_1, w) \wedge p(w, v_2) \quad (5)$$

Additionally, useful axioms such as transitivity are added. If the binary relation is a function, we also add the order axiom, which means that p_{tc} is a total order for nodes reachable from the same

node:

$$\forall v_1, v_2, v_3 \,.\qquad p_{tc}(v_1, v_2) \wedge p_{tc}(v_1, v_3) \rightarrow p_{tc}(v_2, v_3) \vee p_{tc}(v_3, v_2) \vee v_2 = v_3$$

A challenge when working with p_{tc} is proving that a p_{tc} edge does not exists between two nodes. This is because the intended meaning of p_{tc} is a least solution for Eq. (4), which is not expressible in first-order logic. The main observation that helps to alleviate this problem is that the abstract element is finite, and thus transitive closure can be computed on it using a standard reachability algorithm. Any information about the p-edges that can be expressed in the abstract element can be exploited to approximate the p_{tc} predicate. Thus, we maintain a transitive-closure relation on the abstract element, and every time the prover states that an assertion about the p-relation is valid, this transitive-closure relation is updated. Any new information about p_{tc} is stated as an axiom and added to the $\widehat{\gamma}$ formula of the abstract element.

4.3.1 Tracking Changes in Transitive Closure

When the fields of the heap cells are modified during execution of the program, the binary relations representing these fields change. For keeping the analysis precise, the derived predicates defined over transitive closure of these binary predicates need to be accurately updated. There are two approaches for maintaining these transitive-closure relations. First, work in database theory and dynamic complexity (see [7, 11]) give first-order update formulas for the case of unit changes to the binary relation. Second, the parts of the heap in which the transitive closure has not changed can be approximated. We have implemented both approaches and choose to present the second one here.

The basic observation for finding the parts of an abstract element in which the transitive closure has not changed is that if inside a set A, there is no change in the edge relation then any change in the transitive closure within A is because of a path going out of A and then back in. Thus, if we can prove that there is a set of abstract nodes in which the edge relation has not changed, and has either no outgoing edges, or no incoming edges, the transitive closure of the edge between nodes in A has not changed. The analysis in the beginning of this section already gives a way of finding regions for which there are no outgoing (incoming) edges. To find where the edge relation has changed, we add further queries to the theorem prover. The queries are of the form

$$\forall v_1, v_2 \,.\, a_i(v_1) \wedge a_j(v_2) \rightarrow (p(v_1, v_2) \leftrightarrow p'(v_1, v_2))$$

where p is a binary relation that has changed in the operation and for which transitive closure is tracked.

5. Proof Of Concept

We have created a proof-of-concept implementation of the algorithm using a specialized version of the TVLA system [14]. TVLA is a parametric system for performing shape analysis using abstract domains based on canonical abstraction. We modified TVLA to use the algorithm described in this paper for applying transformers. The SPASS theorem prover [23] is used for proving first-order formulas. The Paradox [4] model finder is used to accelerate the convergence process as described in §3.2.1. We use a specialized version of SPASS that supports proving multiple conjectures simultaneously (see [15]). This version of SPASS also supports incremental addition of axioms during execution. The ability to add axioms incrementally is used in two ways. First, every time an assertion is disproved using the model finder, it is removed from consideration by the theorem prover. Additionally, every time new information about the transitive-closure relation is deduced (see §4.3) it is supplied as another axiom to the prover with no need to restart it.

```
List reverse(List x){
  [0] List y = null;
  [1] while (x != NULL){
  [2]    List t = y;
  [3]    y = x;
  [4]    x = x->n;
  [5]    y->n = t;
      }
  [6] assert(AcyclicList(y));
  [7] assert(AllReachable(y));
  [8] return y;
}
```

Figure 7. A procedure that destructively reverses a singly-linked list.

The performance of backward analysis is significantly improved by adding forward information as described in §3.2.2. For most examples, we use a simple forward analysis in which only the cells directly pointed to by pointers are tracked. Aliasing between pointers and direct field references between points (such as immediately following the operation `x->field = y`) can be recovered from the analysis. This allows the backward analysis to limit the possible location of pointers and reduces the number of case splits that FOCUS performs. Some assertions for the `append` and `delete` examples require a more sophisticated (yet still cheap) forward analysis that also tracks reachability from the input variables.

5.1 Programs Analyzed

We have used the system to infer preconditions for several programs. This section describes the programs analyzed and the preconditions recovered. In the experiments, we have analyzed two types of programs: programs that manipulate acyclic singly linked lists and programs that manipulate arrays. The method can be extended to support other data structures by choosing different derived predicates in the same way that TVLA can be extended. For example, see [16] for methods of analyzing cyclic singly linked lists using TVLA.

To clarify, the analysis assumes that the original lists are acyclic and ensures that no cycles are formed during the analysis by adding an extra assertion each time an operation $x.field = y$ is performed to assert that x is not reachable from y.

5.1.1 Reverse

We have run our analysis on a procedure that destructively reverses a singly linked list (see Fig. 7). Most assertions in the procedure do not require a precondition (i.e., the solution for the entry label is the empty abstract set, which means that the precondition is $true$). The only assertion that requires a precondition is the assertion that no memory is leaked by the procedure. The analysis infers that the precondition for this assertion is that there is no element that was not reachable from the head of the list at the beginning of the procedure.

We have also analyzed reverse in a richer context in which the order between the data elements of the list is tracked. We use an extra assertion that all elements are in reverse order at the end of the procedure. The precondition found for this assertion is that the are no two elements in the list that are not in order in the beginning of the procedure.

5.1.2 Append

As described in §2, we have analyzed append and discovered preconditions required for the result to be acyclic and free of memory leaks. To achieve a precise enough precondition, the assertion of no

```
void delete(List x, int delval) {
  [1] List elem = x;
  [2] while (elem != NULL; ) {
  [3]   if (elem->val == delval) {
  [4]     if (prev == NULL)
  [5]       x = elem->n;
  [6]     else
  [7]       prev->n = elem->n;
  [8]     elem->n = NULL;
  [9]     free(elem);
  [10]    break;
        }
  [11]  prev = elem;
  [12]  elem = elem->n;
      }
}
```

Figure 8. A procedure that deletes an element from a singly-linked list.

```
int max(int[] a){
[0] int mi;
[1] int i = 1;
[2] while(i < a.length){
[3]    if (a[mi] > a[i])
[4]       mi = i;
[5]    i++;
    }
[6] assert ∀0≤j<a.length . a[mi]≥a[j];
[7] return mi;
}
```

Figure 9. A faulty procedure for finding the index of a maximal element of an array.

memory leaks requires the use of a forward analysis, that reasons about reachability from the initial pointers.

5.1.3 Delete

The delete procedure (Fig. 8) accepts a singly linked list and a value, and removes the first occurrence of the value in the list. Note that the condition in line `[3]` cannot be modeled directly in our abstract domain and is thus considered as non-deterministic choice. The assertion of no memory leaks requires the use of a forward analysis that reasons about reachability from the initial pointer. The precondition for this procedure is that no memory has leaked before the procedure started.

5.1.4 Strlen

As explained in §2, we have analyzed the procedure `strlen` and found the expected precondition.

5.1.5 Max

The final example is a procedure in Fig. 9. The procedure traverses an array starting from index 1, and searches for the maximal element. The problem with this procedure is that `mi`, which should point to the maximal element, is not properly initialized. The assertions in this example check that no array out-of-bounds accesses occur, and that at the end of the procedure there is no element whose value is greater than the one returned. The analysis is able to show that there are two types of possible problems: (1) The initial value of `mi` is out-of-bounds, and (2) the initial value of `mi` is not 0 and the first element of the array is the maximal element of the array.

6. Related Work

Backwards analysis has a long history, going back to Dijkstra, who introduced weakest preconditions (wp) and weakest liberal preconditions (wlp). In [22] a method for underapproximating wlp was proposed, based on forgetting terms in the invariant. Another possible method is to compute preconditions by combining forward and backward program analysis (e.g., see [6]). This approach can be efficient and precise because it can narrow preconditions based on forward information.

A novel random solution for simultaneously overapproximating strongest postconditions and underapproximating weakest liberal preconditions was presented in [10]. Unfortunately, it is not clear how to apply their solution in our setting.

The nifty trick of underapproximating an abstract value by overapproximating the abstract states that represent errors and then negating the results is rather standard, and has been employed in both the program-analysis (e.g., [2]) and the model-checking communities.

Computing Preconditions for Shape Analysis

Several recent works have tackled the problem of inferring preconditions for programs that manipulate pointers, the heap, and arrays as a method to avoid applying expensive program analysis to the whole code.

Our work is inspired by [3], which establishes preconditions on singly-linked lists to guarantee the absence of memory errors. The answer reported by their precondition-discovery algorithm may be unsound, and thus an extra forward analysis is performed to check if the precondition indeed guarantees the absence of memory errors. In contrast, our algorithm is guaranteed to be sound and can identify preconditions even when their method fails. Moreover, our algorithm can operate on many interesting data structures and prove different properties specified using formulas in first-order logic with transitive closure. Indeed, even for singly-linked list, and for proving the absence of memory errors, our method allows compositional reasoning by identifying a precondition for a given method and propagating the precondition to calls of this method.

A specialized backward algorithm for inferring the absence of memory leaks in acyclic linked lists by static reference counting was presented in [18]. The algorithm also uses the approach of working backward from error states.

In [8], it was proposed to infer preconditions on pointers and arrays using polyhedra. Again their method is unsound and requires an extra forward analysis to check if the precondition indeed guarantees the absence of errors.

Employing Theorem Provers for Abstract Interpretation

The use of theorem provers for abstract interpretation continues to be a long-term research goal (e.g., see [9, 19, 12, 24, 1]). The advantage of using theorem provers is precision and predictability, but the cost of using a theorem prover can be substantial.

To reduce the cost of making a large number of separate but related calls to the theorem prover, the algorithm that we have given in this paper employs the variant of SPASS described in [15], in which related proof obligations that result from multiple conjectures over the same axiom set can be handled simultaneously. This brings down the cost of using resolution-based first-order theorem provers for applying abstract transformers in abstract interpretation.

As part of this work, we observed that TVLA's normal heuristics for computing safe transformers [14] are not precise enough to yield useful information in the examples studied. Indeed, one of the problems unique to backward reasoning is the need to reason symbolically to avoid state-space explosion, and resolution-based methods operate well in these situations.

7. Conclusion

In this paper, we have shown how to harness abstract interpreters and theorem provers to conservatively underapproximate preconditions for programs with destructive updating to fields of dynamically allocated storage. Technically, we infer preconditions that include quantified invariants; these can express shape invariants on an unbounded number of memory locations. Our methods can also infer preconditions for programs that manipulate unbounded-size arrays. A prototype implementation demonstrates that the method is feasible and precise enough.

The key techniques that are used are: (a) Permitting FO(TC) assertions, which allows to define succinctly the set of required assertions and the meaning of atomic statements. (b) Abstractions that provide the necessary distinctions among different memory locations. This amounts to working with a small subset of normalized FO(TC) formulas [25, 1]. One of the most important distinctions is reachability from program variables, which allows our abstractions to discriminate among different segments of data structures. (c) Employing resolution-based theorem provers to convert FO(TC) formulas into abstractions. The use of model finders as external aids, and the ability to incrementally supply the theorem prover with additional axioms, both contribute to making the technique feasible.

References

[1] C. Bouillaguet, V. Kuncak, T. Wies, K. Zee, and M. Rinard. Using first-order theorem provers in the Jahob data structure verification system. In *Verif., Model Checking, and Abs. Interp.*, 2007.

[2] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *Prog. Lang. Design and Impl.*, 1993.

[3] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. Footprint analysis: A shape analysis that discovers preconditions. In *Static Analysis Symp.*, 2007.

[4] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *CADE-19, Workshop W4. Model Computation — Principles, Algorithms, Applications*, 2003.

[5] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Princ. of Prog. Lang.*, 1979.

[6] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3), 1992.

[7] G. Dong and J. Su. Incremental and decremental evaluation of transitive closure by first-order queries. *Inf. & Comput.*, 120, 1995.

[8] N. Dor, M. Rodeh, and S. Sagiv. CSSV: Towards a realistic tool for statically detecting all buffer overflows in C. In *Prog. Lang. Design and Impl.*, 2003.

[9] S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *Computer Aided Verif.*, 1997.

[10] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *Princ. of Prog. Lang.*, 2007.

[11] W. Hesse. *Dynamic Computational Complexity*. PhD thesis, Department of Computer Science, UMass, Amherst, July 2003.

[12] S.K. Lahiri, T. Ball, and B. Cook. Predicate abstraction via symbolic decision procedures. In *Computer Aided Verif.*, 2005.

[13] T. Lev-Ami, N. Immerman, T. W. Reps, M. Sagiv, S. Srivastava, and G. Yorsh. Simulating reachability using first-order logic with applications to verification of linked data structures. In *Conf. on Automated Deduction*, 2005.

[14] T. Lev-Ami and M. Sagiv. TVLA: A system for implementing static analyses. In *Static Analysis Symp.*, 2000.

[15] T. Lev-Ami, C. Weidenbach, T. Reps, and M. Sagiv. Labelled clauses. In *Conf. on Automated Deduction*, 2007.

[16] A. Loginov, T. Reps, and M. Sagiv. Refinement-based verification for possibly-cyclic lists. In *In Program Analysis and Compilation, Theory and Practice: Essays Dedicated to Reinhard Wilhelm*, 2007.

[17] J.M. Morris. Assignment and linked data structures. In M. Broy and G. Schmidt, editors, *Theoretical Foundations of Programming Methodology*. D. Reidel Publishing Co., Boston, MA, 1982.

[18] M. Orlovich and R. Rugina. Memory leak analysis by contradiction. In *Static Analysis Symp.*, 2006.

[19] T. Reps, M. Sagiv, and G. Yorsh. Symbolic implementation of the best transformer. In *Verif., Model Checking, and Abs. Interp.*, 2004.

[20] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Princ. of Prog. Lang.*, 1999.

[21] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *Trans. on Prog. Lang. and Syst.*, 2002.

[22] N. Suzuki and K. Ishihata. Implementation of an array bound checker. In *Princ. of Prog. Lang.*, 1977.

[23] C. Weidenbach, R. Schmidt, T. Hillenbrand, D. Topic, and R. Rusev. SPASS version 3.0. In *Conf. on Automated Deduction*, 2007.

[24] G. Yorsh, T. Reps, and M. Sagiv. Symbolically computing most-precise abstract operations for shape analysis. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[25] G. Yorsh, T.W. Reps, M. Sagiv, and R. Wilhelm. Logical characterizations of heap abstractions. *ACM Trans. Comput. Log.*, 8(1), 2007.